Aspect-Oriented Software Development with Java Aspect Components

Renaud Pawlak [1], Lionel Seinturier [2], Laurence Duchien [3]
Laurent Martelli [4], Fabrice Legond-Aubry [2], Gérard Florin [1]

[1] CNAM, Lab. CEDRIC, 292 rue Saint Martin, 75141 Paris cedex 03, France
[2] Univ. Paris 6, Lab. LIP6, 4 place Jussieu, 75252 Paris cedex 05
[3] Univ. Lille 1, Lab. LIFL, Bâtiment M3, 59655 Villeneuve d'Ascq, France
[4] AOPSYS, 5 rue Brown Séquard, 75015 Paris, France

31st October 2002

Abstract

In the last four years, our research project dealt with separation of concerns for distributed programming environments and applications. This research effort led to the implementation of the Java Aspect Components (JAC) framework for aspect-oriented programming (AOP) in Java. Among the many requirements for distribution, flexibility and adaptability play a stringent role. The high variability of executing conditions (in terms of resources, servers availability, faults, ...) also brings the need for powerful programming paradigms. This led us to develop a dynamic model of AOP which, unlike statically compiled approaches, allows to on-the-fly deploy and undeploy aspects on top of running applications. This model comes with an UML notation and an implementation. An IDE is provided with JAC to support all the development steps of an aspect oriented application, from its design, to its implementation, and to its deployment.

1 Introduction

In order to handle the complexity of software development, separation of concerns [Par72][Dij76] distinguishes between functional and non functional requirements that needs to be addressed in an application. It is assumed that the efficient handling of this issue is a key to software quality and reuse. Nevertheless, one should notice that the frontier between functional and non-functional properties may be moving depending on the application field: features (e.g. time constraints) may be part of the functional requirements in some domains (e.g. real-time control), and of non-functional ones in other domains (e.g. word processing). Object-oriented programming (OOP) is a powerful tool to handle functional decomposition. Still, non-functional properties are specific in the sense that they can not always be decomposed cleanly from functional ones: most of the time they can only be superposed to the original functional decomposition. This leads to the code tangling phenomenon where a concern is scattered into many different locations (i.e. pieces of functional code), making its development, its maintenance, and its reuse difficult. This phenomenon has been isolated in [KLM+97] and led to the development of a new programming style called aspect-oriented programming (AOP). Since then, several tools and compilers have been developed (among them AspectJ [KHH+01]), and closely related techniques have also been improved (among them Hyper/J [OT01] and composition filters [BA01]).

This article presents our programming environment called Java Aspect Components [JAC]. The two main requirements of this framework are to support dynamicity and distribution. Nevertheless, JAC is also a general purpose AOP environment. As this, it comes with a programming model, a design notation and an API. Previous papers described the programming model [PSDF01a] of JAC, its aspect composition mechanism [PSDF01b], the first elements of our UML notation [PDF+02], and the architecture for distribution [PDF+]. This article sums up the main features of JAC and describes in details our UML notation.

Section 2 introduces the programming model of JAC. The UML design notation is described in section 3. Section 4 reports on the architecture of JAC for distribution support. Implementation details and performance

measurements are provided in section 5. Section 6 provides a comparison with other tools and closely related technologies. Finally, section 7 concludes this article.

2 JAC framework & programming model

The JAC framework is based on the notion of containers. Much like in other component frameworks (e.g. EJB [Sun]), a container is a host for software entities. JAC containers host both business component, and non functional component (called aspect component). As we will see later in section 5, when working with centralized environment, the container is simply a customized Java class loader that performs bytecode adaptations to glue the business and aspect components together. Whenever a distribution concern appear in the application, these containers become remotely accessible (either with RMI or CORBA).

Programming model

JAC identifies three different roles involved in the development of an aspect-oriented application. Application programmer: this role is concerned with the core business of the application. S/he implements the software entities coming from the functional decomposition of the problem. Aspect programmer: this role is concerned with the implementation of non functional services. Up to this stage, these services are independent from the ones defined by application programmers. Software integrator: this role puts application and aspect code together. Two important tasks are under the responsibility of this role: pointcut definitions and aspect composition. For these three roles, the programming model of JAC provides the following software artifacts:

1. Base program: this is the set of Java objects that implements the core functionalities of applications. These are regular Java objects. This set of objects is self sufficient and can be run on a JVM (hence, without any aspect).

2. Aspect components: such a component implements a non functional concern that will later on, be woven on a base program. An aspect component defines a crosscut policy (i.e. the methods of the base program whose semantics is modified by the non functional concern) and some aspect methods (advices in AspectJ) that define the semantical modifications. Aspect methods may wrap (execute before and/or after code), replace or extend the semantics of a base method.

3 Design notation

This section describes our UML profile to support the design of aspect with JAC. Stereotypes are proposed to qualify classes implementing a non functional concern (3) and to qualify pointcut relations 3.2. An example using these two concept is given in section 3.3. Section 3.4 goes a step further and draws some similarities between AOP and the use-provide relationship.

3.1 Aspect component classes

Aspect Components are the central point of our AO framework. They are the implementation units that define extra characteristics that crosscut a set of base objects. The key characteristics of JAC is that the base objects that are involved in a crosscut are not necessarily located on a single container. They are defined in classes called Aspect Component classes (AC-classes for short).

An AC-class is tagged with the ≪aspect≫ stereotype. It contains attributes and methods whose semantics differ from regular methods. AC-methods are meant to extend the semantics of regular classes. The extension is performed on well defined implementation points so that these points actually use aspect-services in order to integrate new concerns (e.g. a base class can be made to use a *Cache* interface if the aspect implements some *caching* concern).

Each AC-method defines some code and extents the semantics of some base methods according to a modality defined by a stereotype. The existing stereotypes for an AC-method m follow.

- ≪before≫ m(...): the AC-method m is executed before a given point (to be specified later, see section 3.2) of the base program.
- ≪after≫ m(...): the AC-method m is executed after a given point of the refined program.
- ≪around≫ m(...): a part of the AC-method m is executed before and another part is executed after a given point of the refined program (these two parts are defined within the implementation of m).
- ≪replace≫ m(...): the AC-method m modifies a given point of the extended program implementation by replacing it by the implementation of m.
- ≪role≫ m(...): the AC-method m can be invoked on the objects that are extended by the AC-class; moreover, the AC-method m can access the extended class attributes and the aspect-class attributes.

For instance, figure 1 shows the caching AC-class *Caching* (with the ≪aspect≫ stereotype). As its name suggests it, this AC-class provides a caching extension mechanism. The job of storing and retrieving values from the cache is delegated to the *Cache* (regular) class. The *whenWrite* method of the AC-class *Caching* is tagged with the ≪after≫ stereotype. It will be executed after any base method associated with *whenWrite* in the pointcut definition (see below section 3.2). The *whenRead* method is tagged with ≪around≫. It will be executed before and after some base methods.

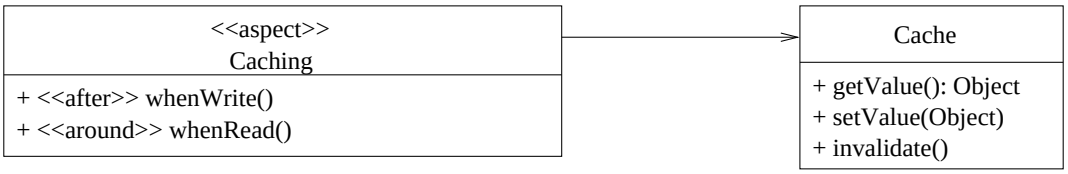

Figure 1: Definition of a caching concern with an AC-class.

3.2 Pointcuts definition

A pointcut relation links an AC-method belonging to an AC-class to a set of elements of a base program. The granularity of the involved elements is the method: pointcuts in JAC can not go deeper than methods to modify the base program semantics (e.g. we can not introspect methods bodies to extend some particular code instructions). Several arguments justify this feature. Firstly, for performance reasons, reifying the whole code structure has a cost which could not be bearable for real-life applications (first experiments with a fully reflective compiler such as OpenJava [Tat99] taught us that). Secondly, extending the semantics of an application requires before that, to understand its original semantics (we can not extend something that is not clearly stated). Most of the time this original semantics is defined through an API, e.g. through methods. So base methods are definitively the best place to perform some semantical extensions.

Two levels of pointcut definition exist with JAC: either the pointcut is defined on a per-class basis, or a per-instance basis.

Class level pointcuts

This level is very much similar to the one found in AspectJ. All the instances of the classes involved in the pointcut are extended by the aspect component. In this case, a pointcut relation is an oriented association from an AC-class towards one or several classes. The association is stereotyped with ≪pointcut≫. The roles have special semantics: they mention which methods of the client class are extended and by which AC-methods. The semantics of the elements mentioned in figure 2 follows:

- a pointcut relation p must go from an AC-class A to a class C (if several classes are involved in the pointcut, several links are drawn between the AC-class and the classes),

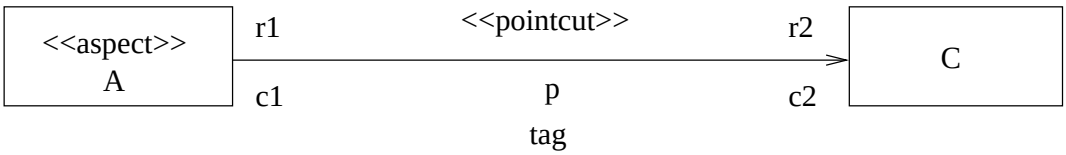

Figure 2: The pointcut association: relating aspects to classes.

Keywords	Semantics
ALL	all the methods
STATICS	all the static methods
CONSTRUCTORS	all the constructors
MODIFIERS	all the methods that modify the object's state, i.e. that modify at least one of the fields
ACCESSORS	all the methods that read the object's state
GETTERS[(...)]	the getters
SETTERS[(...)]	the setters
ADDERS[(...)]	the methods that add an object to a collection
REMOVERS[(...)]	the methods that remove an object to a collection
FIELDGETTERS	all the getters for primitive fields
FIELDSETTERS	all the setters for primitive fields
REFGETTERS	all the reference getters
REFSETTERS	all the reference setters
COLGETTERS	all the collection getters
COLSETTERS	all the collection setters

Table 2: Keywords allowed in pointcut expressions.

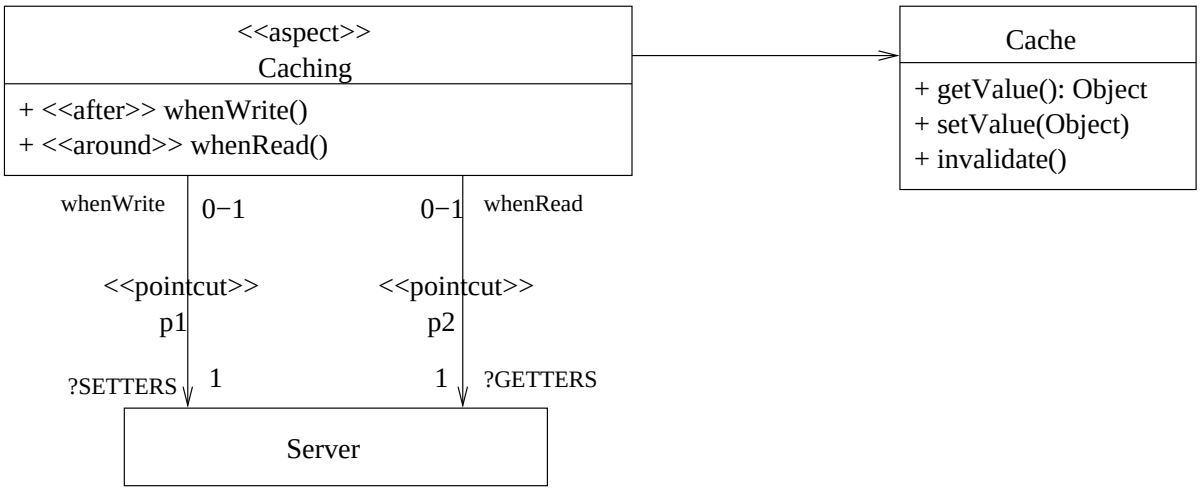

Figure 3: The full caching aspect.

- cardinality *c1* is the number of aspect instances of A that can be in relation with one member of C (default is 0-1),
- cardinality *c2* is the number of members of C that can be in relation with one instance of A (default is * for all),
- role *r1* is the name of an AC-method defined in A that is applied at each base program point denoted by role *r2*,
- role *r2* defines a base program crosscut i.e. a set of joinpoints. *r2* is a logical expression (with AND, OR and NOT operators) where each term is of the form *qualifier methodExpression*.
  - Two mains qualifiers are used: *?* which designated a method execution point, and *!* which designated a method invocation point.
  - *methodExpression* is either a fully-defined method prototype (e.g. *get():int*), or a partially-defined one with GNU-like regular expressions (e.g. *get.*(.*):int* matches all methods whose name starts with *get*, return an integer, and take any parameters), or an expression based on the keyword defined in table 2. For instance, GETTERS(a,b) matches the getter methods for fields a and b. Like in component frameworks such as Java Beans, naming conventions are assumed on method names: getter/setter should be name get/set followed by the field name (starting in upper case). Adders/removers should be named add/remove and take an object as an unique parameter. Each time a new class is loaded in the JAC framework, some introspection and bytecode analysis are performed. A meta-model of the class is constructed on the fly (in a dedicated aspect called RTTI for RunTime Type Information) with annotations that enable to achieve the semantics defined in table 2. For instance, each method bytecode is parsed to determine whether some fields are modified or not. If so, the method is tagged as a MODIFIER in the RTTI aspect. As the process of analyzing the bytecode of many classes can be time consuming, the classes which are never extended by an aspect (i.e. that are simply used by base objects or aspect components), can be excluded from this analysis phase.
- as any UML model element, the pointcut relation can be tagged (*tag*) to express extra semantics that can be used when implementing the model towards a concrete platform; some semantics examples are shown in further sections.

Figure 3 shows two pointcut relations that implement a caching aspect by using the AC-class defined in figure 1. This aspect diagram must be read as follows.

- After the execution of any setter (a method that changes the object state) of a *Server* object, the program must execute the *whenWrite* AC-method.
- Around (i.e. before and after) the execution of any getter (a method that reads the object state) of a server object, the program must execute the *whenRead* AC-method.

Instance level pointcuts

Besides the previously described mechanism, JAC also allows developers to define pointcuts on a per-instance basis. The idea here is to selectively extends the semantics of some instances of a class. The rationale is that in a highly dynamic distributed environment, some server objects may need to customized (e.g. replicated), while others may need to stay unmodified even if they belong to the same class.

One of the difficulty is that, contrary to classes that are straightforwardly named, objects lack any direct naming scheme in Java. The solution taken in JAC is to let the framework attach an unique name to each created instance: the name is the concatenation of the class name in lower case and of an auto-incremented integer (e.g. *server0* designates the first created instance of class *Server*). The framework provides an API to retrieve objects based on their name. This approach is a trade-off between generality and simplicity: it is clear to us that this scheme is usable only if the number of created instances for each class stays small. This scheme also requires the aspect programmer to have a deep understanding of the instance creation process going on in the base program.

To let designers express a per-instance pointcut, aspect component side roles in UML diagram (i.e. *r1* in figure 2) can be extended with an instance name or a regular expression on instance names. For instance *?SETTERS/server0* designates the execution points of the setter methods of instance *server0*, *?GETTERS/server[1-3]* designates the execution points of the getter methods of instances *server1*, *server2* and *server3*.

When distribution comes into play, pointcut definitions can also be filtered based on container names (a container name being a RMI or CORBA URL depending on the chosen communication protocol between JAC remote containers). The idea is to let designers express behaviors that will be dependent on the context into which components are deployed. For instance, one may want to install an authentication aspect only on specific critical hosts, whereas the rest of the application deployed on other hosts stays unmodified, or one may need to install some logging aspect only on a given container. To allow this, pointcut expressions can be extended with container names or regular expressions on container names. Merged with the previous extension for instance names, this leads to a complete scheme where pointcut expressions are of the form: *qualifier methodExpression | instanceExpression | containerExpression* with *instanceExpression* and *containerExpression* being optional. For instance *?ACCESSORS||rmi://myHost/s1* designates the accessors execution points of instances located on JAC container *rmi://myHost/s1*.

3.3 A first simple example

This section illustrates the programming model of JAC based on the *Caching* aspect of figure 3. The details of the API and some tutorials can be found on the JAC web site [JAC].

Figure 4 gives the code of the *Caching* aspect. An aspect component must extend the *jac.core.AspectComponent* class. Among other things this class provides a *pointcut* method that let programmers express a pointcut. The parameters are: the base class this pointcut designates, the *qualifier methodExpression* as a string, the class containing the AC-method, the AC-method involved in the pointcut. Here two such pointcuts are defined. Additional *pointcut* methods are available when *instanceExpression* and *containerExpression* are to be associated to the pointcut.

```
import jac.core.AspectComponent;
import jac.core.Wrapper;
import jac.core.Interaction;

public class Caching extends AspectComponent {
  public Caching() {
    pointcut("Server","?SETTERS",CachingWrapper.class,"whenWrite");
    pointcut("Server","?GETTERS",CachingWrapper.class,"whenRead");
  }

  public class CachingWrapper extends Wrapper {
    private Cache cache = new Cache();
    public void whenWrite( Interaction i ) {
       proceed();
       Object value = i.arg[0];
       cache.setValue(value);
    }
    public Object whenRead( Interaction i ) {
       Object value = cache.getValue();
       if ( value == null )
         value = proceed();
       cache.setValue(value);
       return value;
} } }
```

Figure 4: A simple aspect component implementing a caching concern.

AC-methods are defined in wrapper classes (that extend the *jac.core.Wrapper* class). The following method prototype is mandatory for all AC-methods: they accept only one parameter that is a *jac.core.Interaction* instance. They may return any parameters. The rationale behind this constraint is that AC-methods are upcalled by the JAC framework whenever a call to the base method they extend is issued or executed (i.e. whenever the call matches the pointcut expression). An *Interaction* object i provides data about the current call: arguments of the call (in the *arg* array), a reference to the base object (*i.wrappee*), and some methods to store and retrieve context parameters (for instance, parameters that can be added by an AC-method on the caller side, and that can later on retrieve on the receiver side by another AC-method).

3.4 Extended design notation for distribution

The group paradigm

In the previous caching example, the semantics modification introduced by the caching concern into the application is quite symmetric. Concretely, it means that all the objects that are modified to implement caches (the *Server* objects) can be seen as modified by the same abstract transformation rule. However, one may want to weave the *Caching* aspect to different classes. Thus, another designation mechanism is needed to express the fact that a set of well-defined objects implements the same concern.

This need for a new kind of structured elements brings us to focus on the group notion. If we look at the group notion very carefully, we can notice that it is tightly linked to aspects. Indeed, contrary to a class that abstractly represents a set of instances realizing the same functional characteristics, a group is, in our definition, an abstract representation of a set of instances that do not necessary have homogeneous functional types but that are logically grouped together because they implement the same service (server groups) or use the same one (client groups).

Figure 5 represents the application of the caching aspect on a group of servers that implements the server part of a simple client/server application. We use an instance diagram so that it becomes obvious that the group on the top of the figure is a non-uniform set of instances (the three instances a, b, and c belong to three different classes A, B and C). As shown on this figure, the application of the caching aspect creates a new group that contains instances of a *Cache* class that provides the caching functionality. In other words, we can say that these *Cache* instances belong to a server group that provides a caching functionality for the client group formed by the a, b, and c servers.

A group-based definition of aspects

It appears that the introduction of the caching concern within the original client/server application is abstractly done by the use of the services the *Cache* group interface provides to the servers group. This can be easily represented in UML by using the ≪use≫ relation as represented in figure 6. In the general case, implementing a new concern may require the use of several interfaces. In these cases, several clients can be related to several servers through some ≪use≫ relations.

Finally, a simple but sufficient definition of an aspect within this context is the following.

Definition: an **aspect** is the implementation of one or many *use-provide* relationship(s) between one or many client group(s) and one or many server groups.

The model of figure 6 clearly brings up a use-provide relationship between a client-group (the servers), and a server group (the caches) that defines the group level services *getValue()*, *setValue(Object)* and *invalidate()*. This relationship implementation requires the application to modify the client group member objects implementation to introduce the caching concern within the application. If this concern implementation is modularized (i.e. if the code that implements the caching concern introduction is locally defined), then the implementation technique follows the AOP guidelines and we can call the obtained module an *aspect*.

At the analysis level, to express the fact that a use-provide relationship is implemented in an aspect-oriented fashion, the application designer can add a tagged value "aspect:*aspectName*" to all the ≪use≫ relationships implemented by the aspect called *aspectName* (see figure 6).

Thus group-oriented modeling allows the designer to explicit in a comprehensive way what parts of the (distributed) application are aspects and what parts are not. In fact, for each modeled group level use-provide relationship, aspect-oriented techniques can be used to separate concerns within the final implementation.

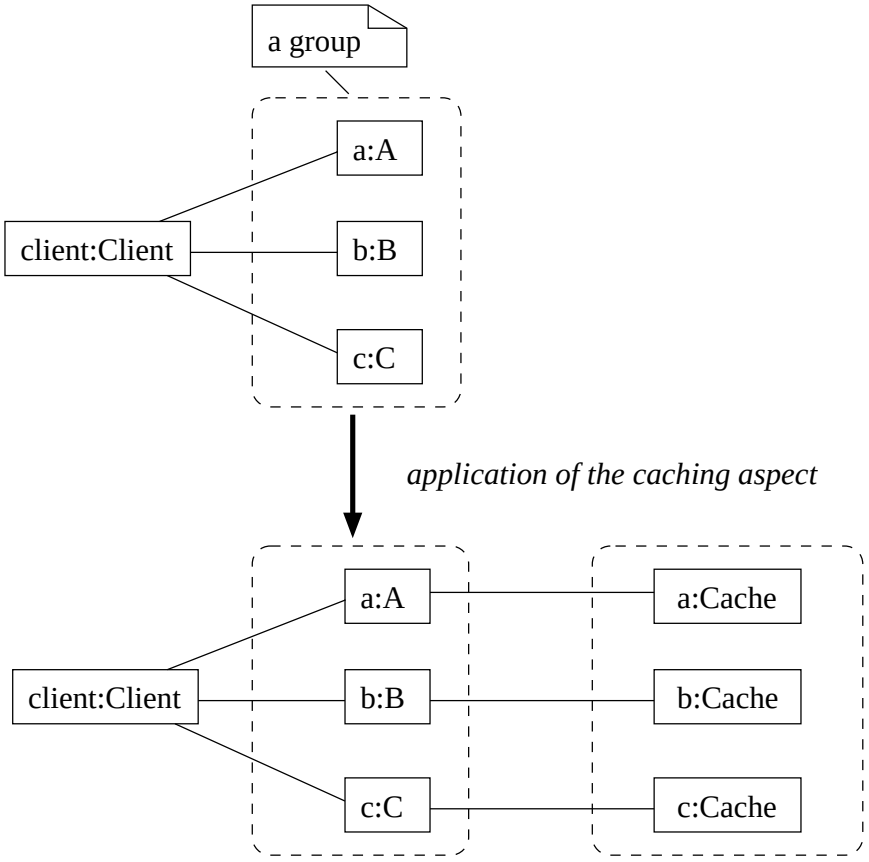

Figure 5: Relating aspects to group.

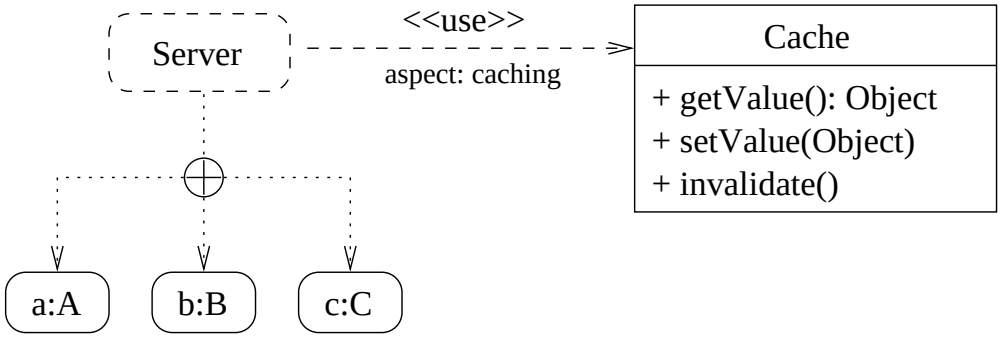

Figure 6: The use relationship between a client group (the base program) and a server group (the aspect program).

Finally, each time the designer encounters the pattern of one or several use-provide relationship between groups, s/he can ask her/himself if an aspect would be well suited in this case. Despite the use of an aspect or not is mainly related to the designer experience and choices, we can give some clues on when an aspect will be better suited than a classical design.

- The client group is heterogenous: it means that the use of the server services are spread all over the client-group member classes. This is by essence a crosscuting concern and some extra design is necessary to cleanly modularize all the involved dependencies (for instance, we can use inheritance that implies bad composition with other concerns or we can use some delegation related pattern which leads to a more complex design model). In this case, the use of an aspect can greatly simplify the programmer's task and will ensure good maintenability and evolutivity of the final implementation, even if some concerns are added afterwards.
- Several homogeneous client groups use the same server group: this is exactly the same situation that above since several homogeneous groups can be modeled into one heterogenous group.
- Several client groups use several server groups but it seems that the final purpose of these use-provide relations enters into one same concern for the final application: this is a more contextual choice that depends on the knowledge of the modeled domain.

Figure sums up the notion introduced in this section and proposes a UML meta model where additions introduced by JAC are drawn with bold lines.

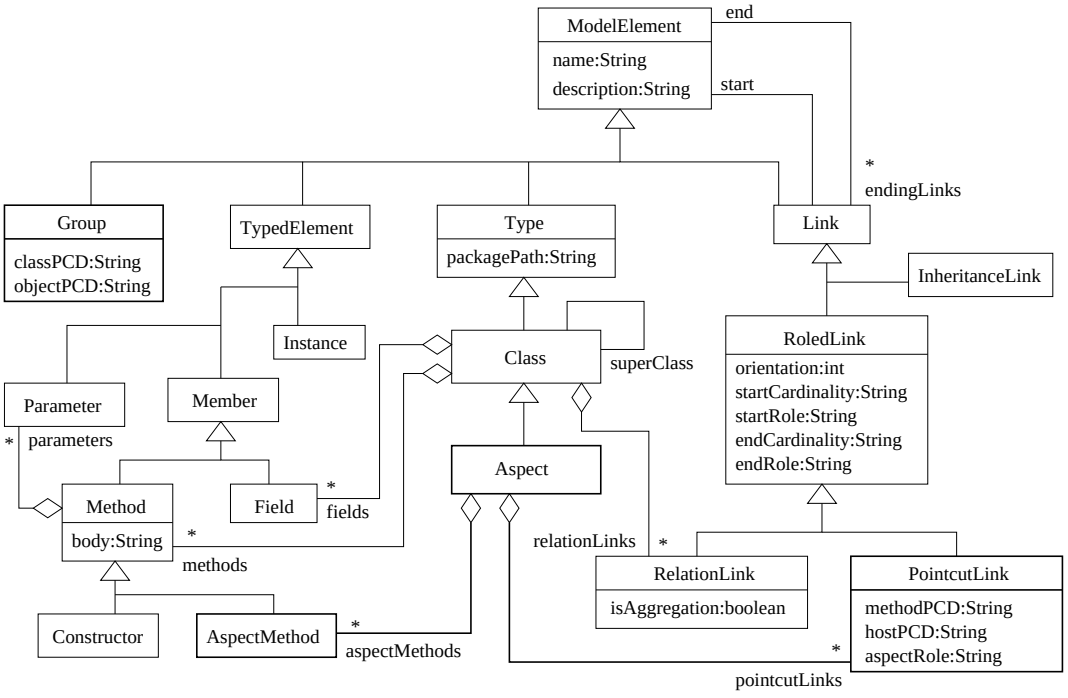

Figure 7: The UML extension meta-model.

4 JAC architecture for distribution

4.1 Aspects deployment and distribution

This section sums up the features of the architecture set up in JAC to handle distribution. This architecture is called AODA (Aspect Oriented Distributed Architecture) and is described in more details in [PDF+].

The AODA provides functionalities to deploy base programs and aspect components.. The idea is to allow the natural and consistent cohabitation between distribution and aspects. To do this, the AODA provides core features to support distributed aspects. Figure 8 shows how the AODA manages distributed aspects. The top of the figure is a simple application composed of a set of components. The middle of the figure shows the same application, but extended by a sample aspect. Finally, the bottom part depicts the application deployed by the AODA. We can notice that each container contains a local instance of the original aspect, hence, the aspect is applied on each container in the same way. The set of containers where the aspect is present is called an aspect-space so that it can finally be regarded as a single but distributed aspect.

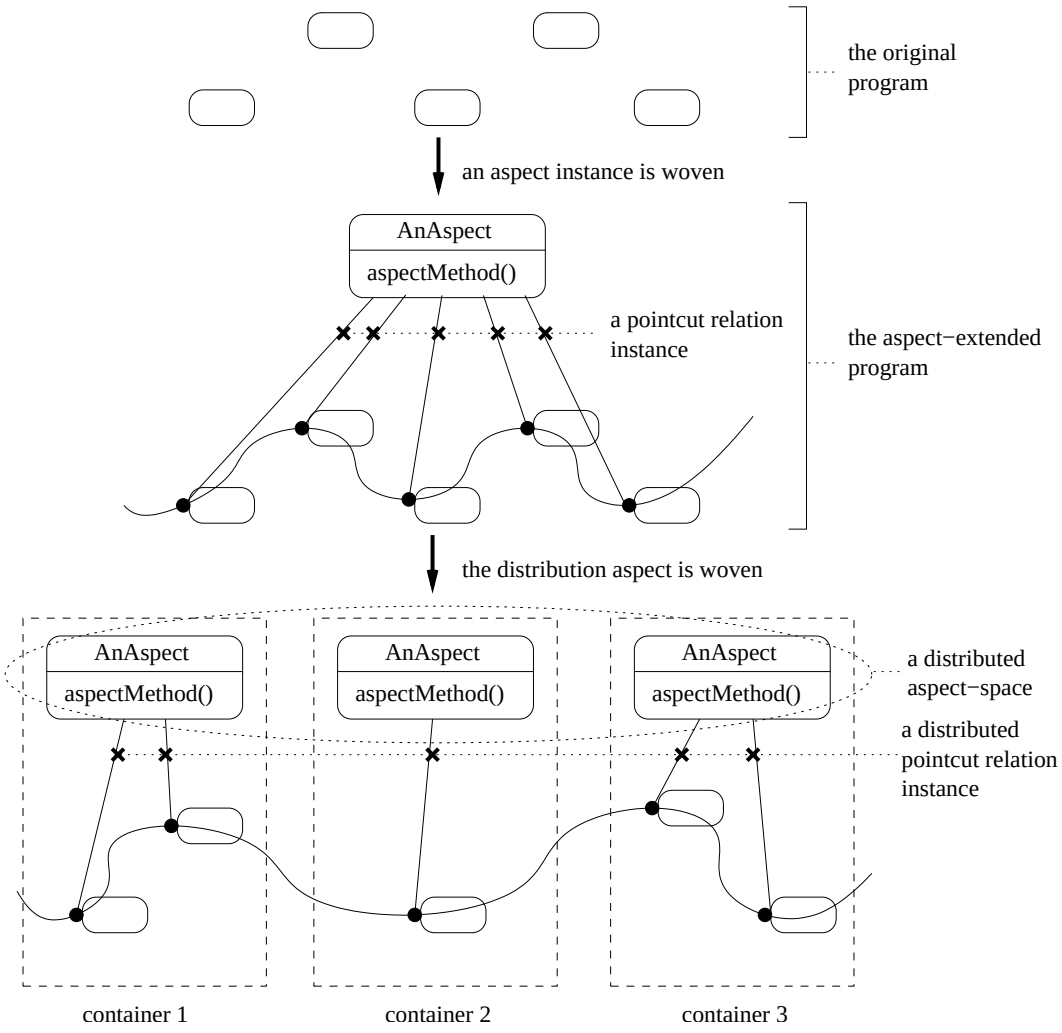

Figure 8: AODA: distributed support of aspects.

Our motivation for distributed aspects support is to allow the aspect programmer to express global and decentralized program properties. Indeed, it happens quite often that a non-functional property crosscuts a set of objects that are not located on the same container. For instance, when adding an authentication concern, the capacities may be checked on several server containers so that it is very useful to modularize all the authentication definition in one unique aspect definition that is seamlessly applied to the whole distributed application.

4.2 Distributed application example

This section presents a simple example of a distributed application with JAC. Readers interested in reading more detailed examples of distributed programming with JAC can refer to [Paw02] where, among other things, a replicated load-balanced server is described.

Let us take the simple example of a three node ring which provides a simple functionality to pass a token between members of a ring. The functional base program is composed of three objects that are the nodes of the ring; So the first step is to develop some base level classes (this code samples can also be found in the JAC version that can be downloaded from [JAC]).

```
public class Ring {
   public static void main( String[] args ) {
      RingElement element0 = new RingElement();
      RingElement element1 = new RingElement();
      RingElement element2 = new RingElement();
      element0.setPrevious( element2 );
      element1.setPrevious( element0 );
      element2.setPrevious( element1 );
      element2.roundTrip( 9 );
}  }
public class RingElement {
   public RingElement previousElement;
   public RingElement() {}
   public RingElement( RingElement previousElement ) {
      this.previousElement = previousElement;
   }
   public void setPrevious( RingElement previousElement ) {
      this.previousElement = previousElement;
   }
   public void roundTrip( int step ) {
      if( step > 0 ) previousElement.roundTrip( step-1 );
}  }
```

For now on, one can develop an aspect component that will deploy the three created objects on JAC containers, or use the existing *DeploymentAC* aspect component provided with JAC. Each aspect component woven to an application can be associated with a configuration file that gives, with a script-like syntax, the steps needed to configure it. Each step corresponds to calling a method of the aspect component. For instance, the following script instructs the instance of *DeploymentAC* woven to the previous base program, to:

1. remotely install (AC-method *deploy*) instances *ringelement0*, *ringelement*1, *ringelement2* on containers bound to, respectively, the RMI name *rmi://host0/s0*, *rmi://host1/s*1, *rmi://host2/s*2.

2. create a client stub (AC-method *createAsynchronousStubsFor*) for *ringelement0* on *s2*, a client stub for *ringelement1* on *s*1, a client stub for *ringelement2* on *s2*. The stub delegates method calls to remote instances. By this way, remote communication details are hidden to ring element objects.

```
deploy "ringelement0" "rmi://host0/s0"
createAsynchronousStubsFor "ringelement0" "rmi://host0/s0" "rmi://host2/s2"
deploy "ringelement1" "rmi://host1/s1"
createAsynchronousStubsFor "ringelement1" "rmi://host1/s1" "rmi://host0/s0"
deploy "ringelement2" "rmi://host2/s2"
createAsynchronousStubsFor "ringelement2" "rmi://host2/s2" "rmi://host1/s1"
```

Figure 9 illustrates the topology generated by this configuration script for the deployment aspect.

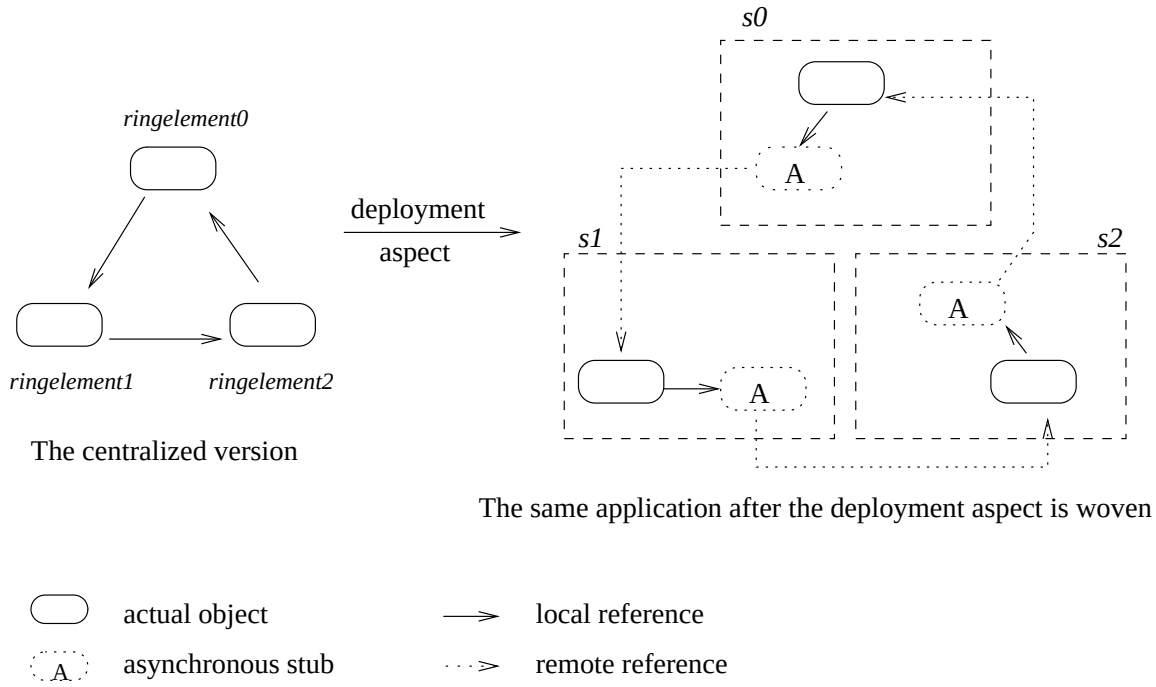

Figure 9: Deployment of the ring application.

Adding a tracing aspect to the ring

Let us assume that we now want to see the round-trip progression on the different containers (in other words, where the token is). Of course, we could modify the *RingElement.roundTrip* method implementation to add a *println* call so that the round-tripping events are logged elsewhere. However, this technique has many drawbacks.

1. It is not dynamic: once the trace handling is there, you must remove it from the code and compile it again to free the components from the trace management.
2. It is not clean: the *RingElement.roundTrip* code is less easy to read for an external eye since it handles a concern that is not purely related to the ring core functionalities.
3. It is less reusable: what happens if you reuse a ring program that has been provided by another programmer and that you do not have the source code? What happens if you want your ring reused? Do you furnish the trace-free version or the traced one?
4. It is not safe: the trace example is simple, but imagine that you introduce a bug or a regression just because you want to add a new technical concern (for instance, you log the traces into a file and, somewhere in all the lines you add into the initial program, you forget to catch the "disk full" or "permission denied" exceptions so that the program stops because of the traces you added). Final users may be upset by this. Using an aspect allows you to modularize the tracing mechanism so that it is much easier to control and to ensure that your modifications do not cause any regression.
5. It is not so simple in a distributed environment: if you want your traces centralized in one unique storage, you may need to install a tracing server... concerns as simple as debugging or logging become more complex when the program is distributed.

For all these reasons and many others, you may want to implement this tracing feature within an aspect. When the application becomes more and more complex, you will take full advantage of this approach. Figure 10 shows the tracing aspect design. The following code is the straightforward JAC implementation of this model.

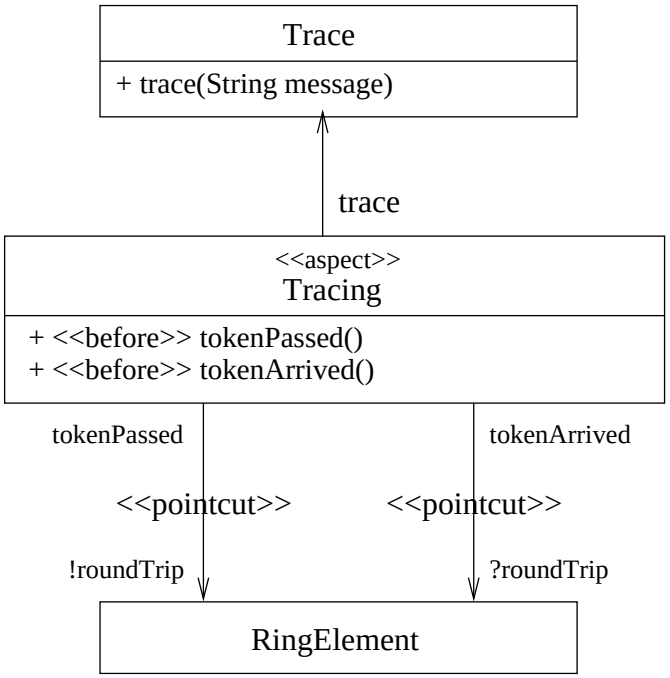

Figure 10: A simple tracing aspect for the ring sample.

```
public class TracingAC extends AspectComponent {
  Trace trace = new Trace();
  TracingAC() {
    pointcut("RingElement", "!roundTrip(int):void",
             TracingAC.TracingWrapper, "tokenPassed");
    pointcut("RingElement", "?roundTrip(int):void",
             TracingAC.TracingWrapper, "tokenArrived");
  }
  class TracingWrapper extends Wrapper {
    public Object tokenPassed( Interaction i ) {
      trace.trace("The token has been passed by "+i.wrappee);
      return proceed();
    }
    public Object tokenArrived( Interaction i ) {
      trace.trace("The token has arrived in "+i.wrappee);
      return proceed();
    }
  }
}
```

As you can see on the figure or within the JAC code, there is no mention of distribution. This means that the aspect also works if the ring is running in a centralized or in a distributed mode (and for any sort of distribution that we provide). By using AODA, we have completely separated the distribution concern from the tracing one but we have also made the aspects and their pointcut semantics inherently distributed. This distributed semantics greatly reinforces the AOP expressiveness by allowing the modularized definition of extensions that crosscut distributed applications.

The way the trace object is actually distributed can be implemented within a deployment aspect (an aspect for the tracing aspect). For instance, if you want all the traces to be centralized on the *s0* container, then just configure a distribution aspect as the follows so that all the calls to the trace features are forwarded on *s0*.

```
deploy "trace0" "s0"
createStubsFor "trace0" "s0" ".*"
```

The final ring application architecture is given in figure 11.

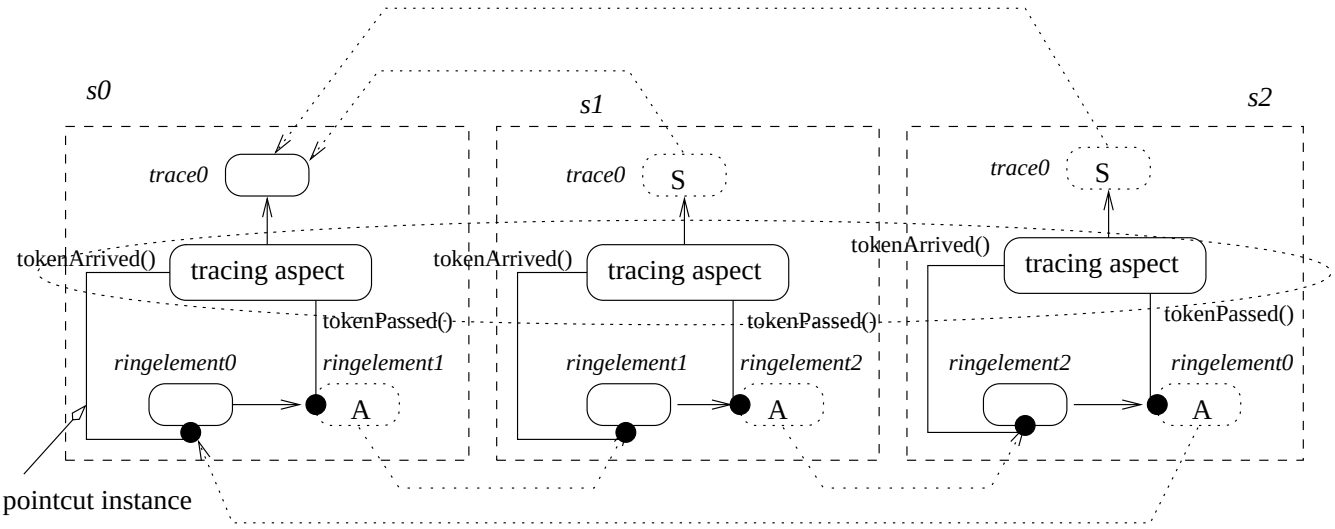

Figure 11: The ring example completed with the distribution and tracing aspects.

5 Implementation & performance issues for JAC

5.1 Implementation of JAC

JAC is entirely written in Java. The aspect weaving is performed at class load time using the bytecode engineering library BCEL. This leaves us the ability to weave existing applications whose source code is not available. In such a case, software integrators need only to know the methods concerned by the pointcut definitions. The current distribution of JAC provides a set of predefined aspects for distribution (either with RMI or CORBA – SOAP to come in future releases), persistence (JDBC or file system), GUI (Swing), authentication, transaction, consistency, load balancing, and broadcasting. A GUI console is provided to on-the-fly weave or unweave aspects on top of a running application. A CASE tool implementing the UML design notation defined in section 2 is also available (see figure 12 for a screenshot).

The joinpoints considered in JAC are method invocations and executions. Hooks are introduced towards woven aspects whenever these events are generated. The idea is not new and has been proposed by many authors (e.g. [Chi95]) to implement MOPs. It consists in introducing a stub method for each method of a base class. These stub methods introductions are done by translating the original classes so that their instances can support aspect weaving.

We investigated several techniques to perform this translation. One of these is to use compile-time reflection by using an open compiler such as OpenJava [Tat99]. OpenJava is very powerful since it allows all kind of code manipulation at compile-time (it takes Java code and produces a translated Java code). However, since it reifies the whole syntax tree of the program it is quite slow and it is not very well suited to our simple problem (we under-use OpenJava for such a simple translation). Another solution for us is to perform the translation at the

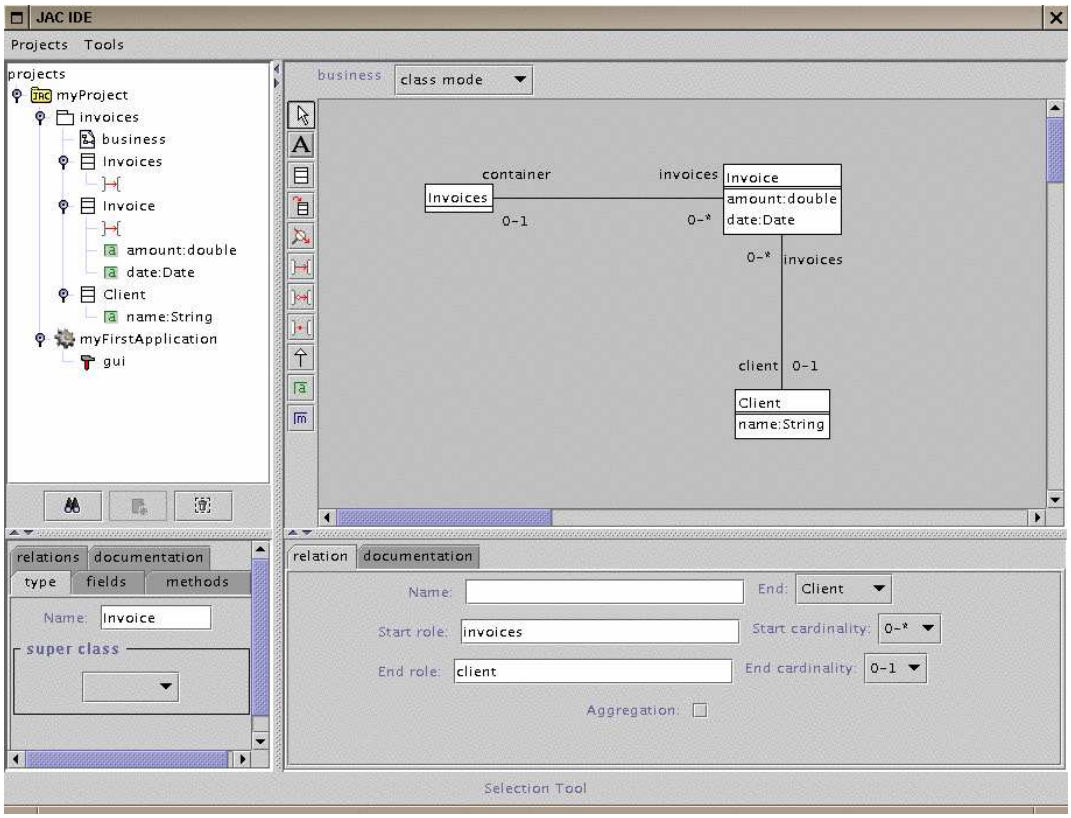

Figure 12: JAC CASE tool screenshot.

bytecode level. Several bytecode translators are available and most of them can work at class load-time. The idea is to use a customized class-loader that reads the class file and modifies the stream contents before actually defining and registering the new class within the JVM. This solution is very well suited for us. First, our translation is very simple and can be performed with very little overhead. Second, we can translate the classes coming from a third-party program or from external libraries with no need of the Java source code.

We implemented the translation with two different bytecode translators. Javassist 1.0 [Chi00] is a reflective high-level translator that hides the complexity of the bytecode format by instantiating a load-time meta model. It is quite fast and easy to use. However, because of its high-level API, Javassist introduces some restrictions on the bytecode manipulations that can be done (for instance, constructors, statics, and method invocations cannot be correctly translated). As a work-around, Javassist 2.0 proposes a low-level API in addition to the high-level one.

BCEL [Dah99] is the most popular bytecode translator. It proposes a very low-level bytecode manipulation interface that makes it very powerful (all kind of translations can be performed). The translation we implemented with BCEL is more complex and slower than with Javassist but the bytecode produced is of better quality.

5.2 Performance measurements

The critical point of the JAC framework in terms of performances is the dynamic wrappers invocation mechanism. Since this invocation relies on reflection in order to achieve dynamic adding or removing of aspects, the performance overhead of JAC mainly comes from the reflective calls overhead. Table 3 shows the performances of empty method calls on regular objects and on JAC wrappable objects. These tests are performed with a bench program that calls several method with different prototypes and that is available in the JAC distribution [JAC]. The bench program was run under Linux with a Pentium III 600 MHz with 256KB of cache and with the SUN's Java HotSpot Client VM version 1.4.

One can see that a call on a JAC wrappable object is comparable to a reflective call on a regular Java object

Type of calls	Number of calls	Total time (ms)	Time per call	Overhead
(A) regular object calls	6,000,000	55	~9.16 ns	-
(B) reflective calls	60,000	47	~0.78 μs	(A)x 85
(C) JAC objects calls (0 wrapper)	60,000	61	~1 μs	(A)x 111 (B)x 1.29
JAC (1 wrapper)	60,000	85	~1.41 μs	(C)+41%
JAC (2 wrappers)	60,000	110	~1.83 μs	(C)+83%
JAC (3 wrappers)	60,000	130	~2.16 μs	(C)+116%

Table 3: Comparative performance measurements for Java and JAC.

(with an overhead of 29%). Each time a wrapper is added, an overhead of about 40% of the initial time is added (note that the bench adds empty wrappers that just call *proceed* in their implementations).

Finally, the price to pay for adaptability is quite high (as for reflection) compared to compiled approaches such as AspectJ. However, with real-word aspects and especially when the application is distributed, this cost becomes negligible compared to the added cost of remote calls. For the moment, the JAC approach is thus more suited for middle grained wrappable objects (only business objects are made wrappable in real-word applications, technical components that need performances are not aspectized) and for distributed and adaptable programming.

6 Related technologies and tools for AOP

This section compares JAC with existing approaches for AOP or closely related technologies.

The composition filter object model (CFOM) [BA01] is an extension to the conventional object model where input and output filters can be defined to handle sending and receiving of messages. This model is implemented for several languages, including Smalltalk, C++ and Java. The latter implementation is an extension to the regular Java syntax where keywords are added to declare, for instance, filters attached to classes. The goals of this model and ours are rather similar: to handle separation of concerns at a meta level. Nevertheless, JAC does not require any language extension.

AspectJ [KHH+01] is a powerful language that provides support for the implementation of crosscutting concerns through pointcuts (collections of principle points in the execution of a program), and advices (method-like structures attached to pointcuts). Precedence rules are defined when more than one advice apply at a join point. In many features (e.g. pointcuts definition) AspectJ has a rich and vast semantics. Nevertheless, we argue that in many cases that we have studied, simple schemes such as the wrapping technique proposed by JAC are sufficient to implement a broad range of solutions dealing with separation of concerns.

Aspectual components [LLM99] and their direct predecessors adaptative plug and play components [ML98, MSL00] define patterns of interaction, called participant graphs (PG), that implement aspects for applications. PGs contain participants roles (e.g. publishers and subscribers in a publish/subscribe interaction model) that, (1) expect features about the classes upon which they will be mapped, (2) may reimplement features, and (3) provide some local features. PGs are then mapped onto class graphs with entities called connectors, that define the way aspects and classes are composed. Aspectual components can be composed by connecting part of the expected interface of one component to part of the provided interface of another. Nevertheless, it seems that by doing so, the definition of the composition crosscuts the definition of the aspects, loosing by this way the expected benefits of AOP.

Subject oriented programming [HO93][OKH+95] (SOP) and its direct successor the Hyper/J tool [TOHS99], provide the ability to handle different subjective perspectives, called subjects, on the problem to model. Subjects can be composed using correspondence rules (specifying the correspondences between classes, methods, fields of different subjects), combination rules (giving the way two subjects can be glued together, and correspondence-and-combination rules that mix both approaches. Prototype implementations of SOP for C++ and Smalltalk exist, and a more recent version for Java called Hyper/J is available. This latter tool implements the notion of hyperspace [OT01] that *permits the explicit identification of any concerns of importance.*

An approach relatively close to the spirit of JAC is the Mozart project [Van99]. Mozart is an open distributed programming system on the Oz language and uses object/component-orientation, declarative, logic, and constraint programming to support the separation of the functional and of the distribution concerns. It provides a good separation of concerns degree with a support for multiple paradigms. Despite its complete nature, the core Mozart does not take the full advantage of new AO programming concepts such as aspect-classes or pointcuts. In our opinion, it is therefore more difficult to apprehend and less flexible since the provided concerns are built-in (but configurable) within the system.

Superimpositions [SK02] are also an approach for separation of concerns in distributed environments. It is a theoretical work that furnishes a language that can be applied to AOP. We are currently working on using some of the fundamental concepts of superimpositions in our aspects.

Finally, several projects such as Lasagne [TVJ+01], JMangler [KCA01], or PROSE [PGA02] provide dynamic weaving/unweaving of aspects that makes them close from the JAC implementation. However, none of them fully handle the automatic distribution of the aspects when the application is distributed.

7 Conclusion

JAC is a framework for aspect-oriented programming (AOP) in Java. It provides a general programming model and a number of artifacts to let programmers develop aspect-oriented applications in a regular Java syntax (i.e. without any syntactical language extensions). The main elements managed by the framework are aspect components (AC for short). They are the piece of code that capture a cross-cutting concern. Like in others AOP approaches, the idea is to modularize this concern to ease its maintenance and its evolution. JAC provides containers to host AC and business (also called base) components. In the current version of JAC (downloadable from our web site [JAC]), these containers are remotely accessible either with RMI, or with CORBA. Further developments are underway for the SOAP communication protocol.

ACs define two main elements: pointcut relations and AC-methods. Pointcut relations are the methods of the base program whose semantics is meant to be extended by the AC. AC-methods are the blocks of code that perform the extension. The originality of pointcut relations with JAC is that they can be defined on a per-class basis (all instances of some given classes are equally extended), or on a per-instance basis (only given instances of some classes are extended). To achieve this feature, a naming scheme is provided for each base instance managed by the framework. A language for pointcuts definition is provided that let developers filter instances based on their name or on the name of the container hosting them. The AC-methods provide code that can be run before and/or after, or replace the methods designated by the pointcut.

An UML notation has been proposed in section 3. The stereotypes provided enable designers to express all the above mentioned elements concerning AC, pointcut relations, and AC-method. Section 3.4 investigated some more advanced concepts where AOP is compared to the use-provide relationship and to some notions of groups of heterogeneous classes. The notation is supported by a CASE tool.

References

[BA01] L. Bergmans and M. Aksit. *Software Architectures and Component Technology*, chapter Constructing Reusable Components with Multiple Concerns Using Composition Filters. Kluwer Academic Publishers, 2001.

[Chi95] S. Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 285–299. ACM Press, October 1995.

[Chi00] S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer, June 2000.

[Dah99] M. Dahm. Byte code engineering. In *Proceedings of JIT'99*, 1999. `http://bcel.sourceforge.net`.

[Dij76] E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings of OOPSLA'93*, volume 28 of *SIGPLAN Notices*, pages 411–428. ACM Press, October 1993.

[JAC] The JAC project home page. `http://jac.aopsys.com`.

[KCA01] G. Kniesel, P. Constanza, and M. Austermann. JMangler - a framework for load-time transformation of java class files. In *Proceedings of the IEEE Workshop on Source Code Analysis and Manipulation (SCAM'01)*. IEEE Computer Society Press, November 2001.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[LLM99] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University's College of Computer Science, April 1999.

[ML98] M. Mezini and K. Lieberherr. Adaptative plug-and-play components for evolutionary software development. In *Proceedings of OOPSLA'98*, volume 33 of *SIGPLAN Notices*, pages 96–116. ACM Press, 1998.

[MSL00] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In L. Bergmans and M. Aksit, editors, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000.

[OKH+95] H. Ossher, K. Kaplan, W. Harrison, A. Matz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA'95*, volume 30 of *SIGPLAN Notices*, pages 235–250. ACM Press, 1995.

[OT01] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001.

[Par72] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Paw02] R. Pawlak. AOSD with JAC - chapter 8 from PhD. thesis. CNAM Paris, December 2002.

[PDF+] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. JAC: An aspect-based distributed dynamic framework. Submitted to British Computer Science.

[PDF+02] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli. An UML notation for aspect-oriented software design. In *Workshop on Aspect-Oriented Modeling with UML at AOSD'02*, April 2002.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, 2002.

[PSDF01a] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic wrappers: Handling the composition issue with JAC. In *Proceedings of TOOLS USA 2001*. IEEE Computer Society Press, July 2001.

[PSDF01b] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection'01*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, September 2001.

[SK02] M. Sihman and S. Katz. A calculus of superimpositions for distributed system. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 28–40, 2002.

[Sun] Sun Microsystems. *Enterprise Java Beans.* `http://www.javasoft.com/products/ejb`.

[Tat99] M. Tatsubori. An extension mechanism for the Java language. Master of Engineering Dissertation, Graduate School of Engineering, University of Tsukuba, Ibaraki, Japan, February 1999. `http://www.csg.is.titech.ac.jp/openjava/`.

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[TVJ+01] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, Joergensen, and N. Bo. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE'01*, 2001.

[Van99] Peter Van Roy. *On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart.* World Scientific, Tohoku University, Sendai, Japan, July 1999.